你真的了解 package.json 么

在 Node.js 中，模块是一个库或框架，也是一个 Node.js 项目。Node.js 项目遵循模块化的架构，当我们创建了一个 Node.js 项目，意味着创建了一个模块，这个模块的描述文件，被称为 package.json。

我之前看别人项目中 package.json 文件的 scripts 这样写：

```
"dev": "rimraf \"config/.conf.json\" && rimraf \"src/next.config.js\" && cpx \".conf.json\" \"config/\" && nodemon server/index.ts",
"clean": "rimraf ./dist && mkdir dist",
"prebuild": "npm run clean",
"build": "cross-env NODE_ENV=production webpack"
```

bin

它是一个命令名和本地文件名的映射。在安装时，如果是全局安装，npm 将会使用符号链接把这些文件链接到 prefix/bin，如果是本地安装，会链接到./node_modules/.bin/。

通俗点理解就是我们全局安装， 我们就可以在命令行中执行这个文件， 本地安装我们可以在当前工程目录的命令行中执行该文件。

```
"bin": {
    "gynpm": "./bin/index.js"
}
```

要注意： 这个 index.js 文件的头部必须有这个#!/usr/bin/env node 节点，否则脚本将在没有节点可执行文件的情况下启动。

小实验

通过 npm init -y 创建一个 package.json 文件。

```
{
    "name": "cc",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "bin": {
        "mason": "./index.js"
```

```
    },
    "scripts": {},
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {}
}
```

在 package.json 的同级目录新建 index.js 文件

```
#!/usr/bin/env node

console.log('cool')
```

然后在项目目录下执行： mac 下： sudo npm i -g, window 下： npm i -g

接下来你在任意目录新开一个命令行， 输入 mason， 你讲看到

cool 字段。

不知道通过这个小实验能不能帮助大家更好的理解这个 bin 的作用。像我们常用的 vue-cli， create-react-app 等都是通过 bin 属性将命令映射到了全局上。

main

main 很重要， 它是我们项目的主要入口。

```
"main": "app.js"
```

像这样， 我们项目就会以根目录下的 app.js 文件作为我们的项目入口文件了。

scripts

npm 允许在 package.json 文件里面，使用 scripts 字段定义脚本命令。 优点： 项目的相关脚本，可以集中在一个地方。

不同项目的脚本命令，只要功能相同，就可以有同样的对外接口。用户不需要知道怎么测试你的项目，只要运行 npm run test 即可。

可以利用 npm 提供的很多辅助功能。

npm 脚本的原理非常简单。每当执行 npm run，就会自动新建一个 Shell，在这个 Shell 里面执行指定的脚本命令。因此，只要是 Shell（一般是 Bash）可以运行的命令，就可以写在 npm 脚本里面。

比较特别的是，npm run 新建的这个 Shell，会将当前目录的 node_modules/.bin 子目录加入 PATH 变量，执行结束后，再将 PATH 变量恢复原样。

这意味着，当前目录的 node_modules/.bin 子目录里面的所有脚本，都可以直接用脚本名调用，而不必加上路径。比如，当前项目的依赖里面有 Mocha，只要直接写 mocha test 就可以了。

*通配符

*表示任意文件名，**表示任意一层子目录。

```
"lint": "jshint *.js"
"lint": "jshint **/*.js"
```

如果要将通配符传入原始命令，防止被 Shell 转义，要将星号转义。

```
"test": "tap test/\*.js"
```

脚本传参符号： --

```
"server": "webpack-dev-server --mode=development --open --iframe=true
",
```

脚本执行顺序

并行执行（即同时的平行执行），可以使用&符号

```
$ npm run script1.js & npm run script2.js
```

继发执行（即只有前一个任务成功，才执行下一个任务），可以使用&&符号

```
$ npm run script1.js && npm run script2.js
```

脚本钩子

npm 脚本有 pre 和 post 两个钩子， 可以在这两个钩子里面，完成一些准备工作和清理工作

eg:

```
"clean": "rimraf ./dist && mkdir dist",
"prebuild": "npm run clean",
"build": "cross-env NODE_ENV=production webpack"
```

npm 默认提供下面这些钩子:

```
prepublish, postpublish
preinstall, postinstall
preuninstall, postuninstall
preversion, postversion
pretest, posttest
prestop, poststop
prestart, poststart
prerestart, postrestart
```

拿到 package.json 的变量

npm 脚本有一个非常强大的功能，就是可以使用 npm 的内部变量。

首先，通过 npm_package_前缀，npm 脚本可以拿到 package.json 里面的字段。比如，下面是一个 package.json。

```
// package.json
{
  "name": "foo",
  "version": "1.2.5",
  "scripts": {
    "view": "node view.js"
  }
```

```
}
```

我们可以在自己的 js 中这样：

```
console.log(process.env.npm_package_name); // foo
console.log(process.env.npm_package_version); // 1.2.5
```

常用脚本

```
// 删除目录
"clean": "rimraf dist/*",

// 本地搭建一个 HTTP 服务
"serve": "http-server -p 9090 dist/",

// 打开浏览器
"open:dev": "opener http://localhost:9090",

// 实时刷新
 "livereload": "live-reload --port 9091 dist/",

// 构建 HTML 文件
"build:html": "jade index.jade > dist/index.html",

// 只要 CSS 文件有变动，就重新执行构建
"watch:css": "watch 'npm run build:css' assets/styles/",

// 只要 HTML 文件有变动，就重新执行构建
"watch:html": "watch 'npm run build:html' assets/html",

// 部署到 Amazon S3
"deploy:prod": "s3-cli sync ./dist/ s3://example-com/prod-site/",

// 构建 favicon
"build:favicon": "node scripts/favicon.js",
```

介绍几个在 npm 脚本中好用的模块

cpx 全局复制

一个很好用的模块， 可以监视全局文件变化， 并将其复制到我们想要的目录

我们使用 npm 安装就可以在 npm 的脚本中使用了：

"copy": "cpx \".conf.json\" \"config/\" "

这样我们运行 npm run copy 就可以将根目录下的.conf.json 文件拷贝到 config 文件夹下了，如果没有 config 文件夹就会新建一个。

cpx "src/**/*.{html,png,jpg}" app --watch

当 src 目录下的任意.html，.png，.jpg 等文件发生变化就拷贝到 app 目录下。

cross-env 能跨平台地设置及使用环境变量

大多数情况下，在 windows 平台下使用类似于: NODE_ENV=production 的命令行指令会卡住，windows 平台与 POSIX 在使用命令行时有许多区别（例如在 POSIX，使用$ENV_VAR,在 windows，使用%ENV_VAR%。。。）

cross-env 让这一切变得简单，不同平台使用唯一指令，无需担心跨平台问题:

"start": "cross-env NODE_ENV=production node server/index.js",

dependencies 和 devDependencies

这两个主要就是存放我们项目依赖的库的地方了， devDependencies 主要是存放用于本地开发的， dependencies 会在我们开发的时候带到线上。

通过 npm i xxx -S 会放在 dependencies， npm i xxx -D 会放在 devDependencies。所以我们在装包的时候一定要考虑这个包在线上是否用的到， 不要全都放到 dependencies 中，增加我们打包的体积和效率。

peerDependencies

我们在一些项目的 package.json 中看到这个属性， 它主要是考虑兼容问题，通俗点理解， 我们通过这个属性可以告诉要使用我们这个模块的人：

你要使用我， 最好把 xxx1, xxx2 也带上， 不然我可能会给你带来麻烦的。

```
"peerDependencies": {
        "xxx1": "1.0.0",
        "xxx2": "1.0.0",
    }
```

这样在装包的时候同时也会， 带上 xxx1 和 xxx2 这两个包。

个人觉得比较重要的就是这几个了。还有一些， 像： author, version, keywords, description 这些就很好理解了。